8K OMNIMON (TM): L & U
USER'S GUIDE

by

David Young, CDY Consulting

This document covers the additional features of OMNIMON L&U which are not available in the standard version. Please refer to the OMNIMON USER'S GUIDE for descriptions of the standard features. Also, it is possible to use OMNIMONL independently of OMNIMONU. If you are doing so, ignore the descriptions of the commands which only exist in OMNIMONU (N and Y).

Installation

The 8K version of OMNIMON comes in a special 8K chip which has a switch attached to it. Note the orientation of the old OMNIMON chip (the one with the orange label), carefully unplug it by gently prying with a screwdriver (be careful not to gouge the etch underneath), and plug in the new chip in the same orientation. **Do not plug the chip in backwards!** The end with the pins bent under should butt up against the wire connector. If there is some interference, shave a little bit of the plastic off with a sharp knife so that the chip will sit down flush. Now mount the switch somewhere by drilling an appropriate size hole. Since the switch should be convenient to your left hand, mounting it just above the ESC key is recommended although this requires that you lead the wires out the bottom of card cage. Mounting the switch on the left side of the lid would allow you to remove the OS board with the lid, but it requires cutting a hole in the RF shield. Such is the price for the added power of 8K OMNI!

Overview of 8K OMNIMON

All versions of OMNIMON reside in the 4K block of memory at $C000. Implementation of the 8K OMNIMON is done with two 4K banks, only one of which can be active at any given time. The lower 4K (OMNIMONL) contains primarily disk I/O commands while most of the debugging commands reside in the upper 4K (OMNIMONU). Bank selection is achieved via a manual toggle switch.

It is not necessary to memorize which commands reside in each bank because the monitor will tell you to flip the switch if you use a command which is not in the currently selected bank. It will do so in a strange way, but one which took very little code space to implement: all the characters on the screen will turn upside down. When this happens, don't get excited. Just flip the switch and hit RETURN. The command parser will turn the characters right side up again (what a relief!), search the other bank for the command, and execute it if it is found. What if it does not find the command in the other bank either? That's right: the characters will stand on their heads again. If you happen to try a command which is not in either bank, the characters will

remain inverted no matter how many times you flip the switch and hit RETURN. To get out of this infinite loop just hit any key other than RETURN (like the space bar) and the command will be aborted.

A word of warning: **DO NOT FLIP THE SWITCH UNLESS THE CHARACTERS ARE INVERTED!** Otherwise you may crash the system. The reason for this is that bank switching while the CPU is operating out of that bank will not work gracefully unless the code is identical in both banks. Only when the characters are inverted is the CPU operating in a region where the code is identical in both banks. Thus, even if you know that the command is in the other bank, go ahead a type the command and flip the switch only when the characters turn upside down. A little annoying at first, you will soon get used to it, especially when you start to appreciate some of the extra facilities that 8K has to offer.

Choosing which commands were to reside in each bank was a tradeoff between several factors:

1) Grouping the commands to minimize bank switching.
2) Grouping certains commands in the same bank so that they could share common resources.
3) Making the best use of the available space.

It was found that these requirements were fairly complementary: each command seemed to fall naturally into one bank or the other. Most of the disk I/O commands (R,W,L,G,B) are in the lower bank while most of the debugging commands (C,E,J,N,X,Y) are in the upper bank. Some of the most commonly used commands (D,A,T,P) are put in both banks.

You will find that the 'U' bank is somewhat secondary to the 'L' bank. In fact, OMNIMONL will work well without OMNIMONU but the reverse is not true. The following features of the monitor will work only if the 'L' bank is selected:

1) Entering the monitor with OPTION/RESET or SELECT/RESET
2) Exiting the monitor with START/RETURN
3) Making use of the extendability features (fixed entry points)
4) Using the resident Ramdisk handlers

Thus, you should leave the lower bank selected whenever you are not in the monitor.

There are some very powerful features in 8K OMNIMON, especially the combination of binary load (G), user extendability (U) and programmability (F,O). It will be very interesting to find out all the interesting applications people will come up with. If you come up with something nifty, feel free to send it in to CDY for possible inclusion in an OMNIMON User Extensions disk. If your submission is accepted you will be given credit for your work and you will receive a copy of the disk when it is available. Others may receive a copy of the disk by sending $15.00 to CDY. Call for availability.

Assembler: Y addr instruction

This will not take the place of your 2 pass assembler. 'Y' is a line assembler, meaning that you type in a line of assembly language and it will convert it to machine code immediately. If you have ever had to hand patch 6502 object code you will know how handy this function can be. It saves you from having to look up the opcodes in a table. Also, sometimes it is desirable to write a quicky routine to try something out. For example, say you wanted to see what the key codes are that get stored in location $2FC whenever you press a key. To do so you could type in the following commands:

```
>Y 600 LDA $2FC    (get key code)
       CMP #$FF
       BEQ $600    (no key pressed?)
       STA $610    (store new key code)
       RTS
(just hit RETURN to terminate assembler)
```

Notice that you need to type 'Y addr' only once. From that point on the assembler will prompt you for the next instruction. To execute this routine:

```
>J 600
>(START/RETURN)
(press a key)
>D 610    (to see what the key code is)
```

Here are the rules of the assembler:

1) To enter the assembler, type 'Y addr instruction' where 'addr' is the starting address in hex and 'instruction' is a legal 6502 assembly language instruction.

2) To exit the assembler just type RETURN when prompted for the next instruction.

3) If you want to make a change to an assembly language instruction which is already on the screen, just move the cursor up to that instruction (it must be the line with the 'Y' at the beginning), make the change, and hit RETURN. This is valid whether or not you are currently in the assembler.

4) When specifying operands, hex numbers are preceded by '$' and decimal numbers are not.

5) Branch instructions (BEQ, BNE, etc.) are handled quite easily. The operand can be specified as an absolute address (in hex or decimal) and the assembler will calculate the displacement. Or, if you prefer, you can specify a displacement preceded by + or -. Thus, in the example above, the 'BEQ $600' could be replaced by 'BEQ -5' with the same result.

Binary Load / Directory: G (file spec) (addr)

The versatility and convenience of this command is really quite amazing. First of all, it will load any binary load file from a single or double density ATARI DOS compatible disk (2.0S, 2.0P, MYDOS, OSA+2.0, etc.). This includes files that even DOS cannot load because they load on top of it. Secondly, as it searches the disk directory for the specified file, it prints out the filenames, file sizes and start sectors in hex. For example, put a disk with a binary load file into drive #1. Now type:

>G D:filename

Pretty nice. Here are the rules for using 'G':

1) The file specification is similar in format to that of DOS. For example, 'D:FILE', '1:FILE', and 'FILE' all are equivalent to 'D1:FILE'.

2) If you do not give a file specification, 'G' will search the directory and not find a match but in the process will print out the entire directory. Thus 'G(RETURN)' will give the directory of drive #1 while 'G2:(RETURN)' yields the directory of drive #2.

3) Another nice feature of 'G' is that it will print out the load vectors of a binary load file if you hold down the OPTION key while the file is being loaded. It will print out each load vector and pause before it loads the data to satisfy that load vector. Pressing SELECT will load the data and print the next load vector. If you wish to terminate the load, press START. Try loading several binary load vectors while holding down the OPTION key and you will be surprised at how complicated some of them are (the ATARI Macro Assembler for instance).

4) One final option concerns the sector buffer address which 'G' uses while it is loading the file. If you do not specify an address, $400 is the default. If the binary file happens to load into this area ($400-47F for single density or $400-$4FF for double), you may specify another buffer address in hex which does not interfere with the load.

By the way, once the binary load has started, 'G' uses zero page locations $43-49. These are locations reserved for use by DOS, in particular during a binary load. Thus, 'G' should load anything that the 'L' option of DOS does. However, some files may not load correctly if you use the console switches as described in rule #3 above because the resources used to print out the load vectors may interfere with the load itself.

Binary Load Files

This is one area that most people are a little fuzzy on. It's not surprising really, since there does not appear to be any definitive documentation on it anywhere! An exhaustive

presentation will be made here even though it will be quite short. You will see that binary load files are simple but very powerful.

Definition: **load vector** - from 4 to 6 bytes consisting of 2 optional bytes of FF FF, a 2 byte start address, and a 2 byte end address (in that order).

Example: FF FF ØØ Ø6 Ø2 Ø6 - The first 2 bytes are optional and ignored during the load process. The second 2 bytes are a start address of $6ØØ and the last 2 bytes are an end address of $6Ø2.

The only time that the first 2 bytes of FF FF are required is at the beginning of a binary load file. If those bytes are not there, DOS will refuse to perform the binary load. (The 'G' command of OMNIMONL will, however, load it gladly.) The rest of the time the first 2 bytes of FF FF, if they exist, are ignored. The only time that these 2 optional bytes should occur anywhere else but at the beginning of a file is if 2 binary load files were appended together.

What does a load vector do? It tells DOS (or the 'G' command of OMNIMON) where in memory to put the 1 or more bytes which follow in the file. How many bytes is determined by subtracting the start address from the end address and adding 1. In the example above, 3 bytes would be read from the file and put in locations $6ØØ to $6Ø2.

What happens when enough bytes have been read in to satisfy a load vector? These things will happen in this order:

1) Locations $2E2 and $2E3 will be examined. If they are both zero, goto step 2. If they are nonzero, a JSR will be made to the address contained in these locations. Upon return, zero $2E2 and $2E3 and fall into step 2.
2) If the end of file is not reached (i.e., there are more bytes in the file), another load vector is assumed to immediately follow and will be processed as previously described.
3) If the end of file (EOF) is reached, examine locations $2EØ and $2E1. If they are zero, terminate the binary load. If they are nonzero, do a JSR to the address in these locations. Upon return, terminate the load.

Still confused? Let me try to simplify. If you see a load vector like 'E2 Ø2 E3 Ø2', you know that the subroutine at the address specified in the following 2 bytes will get executed immediately, prior to continuing the load process. If you see a load vector like 'EØ Ø2 E1 Ø2', you know that the subroutine at the address in the following 2 bytes will be executed after the end of file is reached.

I hope that this helps you understand this topic a little better. As much as I enjoy talking to people on the phone, I get a little tired of explaining how a binary load file works!

Fill Program Buffer: F addr

If you find yourself using a certain sequence of OMNIMON commands frequently, you may want to program the monitor to execute them automatically. Likewise, if you are debugging or analyzing a program, you may want the monitor to remember the sequence of commands so that you can duplicate them at a later time. The 'F addr' command tells OMNIMON to start saving all your commands at the specified address (the program buffer). It will continue to save your commands until you execute an 'FØ' to tell it to quit. Then the 'O addr' command is used to execute from the program buffer at any later time.

The format of the 'F' command is as follows:

F addr - 'addr' can be any non-ZPAGE address in RAM. It will enter the program mode at this point, remembering all subsequent commands. An addr of Ø terminates the program mode.

The program buffer should be a place in memory that will not interfere with anything else you are doing. As long as you are in the program mode the ASCII data stored in the buffer will continue to grow, eventually wiping out everything in its path if you forget to terminate the program mode with 'FØ'.

Since the data stored in the program buffer is ASCII text, you may edit it if you wish. Just remember that the program data must be terminated with a hex Ø. Also, if you wish to enter the program mode and append to the end of the current data rather start over, you may do so as follows:

```
>T                (to get into character mode)
>S addr FØ        (addr=program buffer; find 'FØ')
A xxxx FØ-------(which previously exited program mode)
>F xxxx           (start filling buffer at that location)
```

One slight annoyance you will encounter in the program mode is that the result of the 'T' command depends on the the current mode (char or hex). If you use the 'T' command as part of the programmed sequence you will want to force one mode or the other at the beginning of the sequence so that you will always get the same results no matter what mode you happen to be in when you execute the sequence later. It is possible to force the hex mode with the following sequence:'A98 Ø'.

If the data of your command buffer is of any importance you may want to back it up occasionally to a scratch disk. This could be done as follows:

1) Find the 'FØ' at the end of the buffer as in the example above and add 3 to determine the end address.
2) Subtract the start address from the end address and divide by the sector size to determine the number of sectors to write.
3) Write that number of sectors off to a scratch disk with the 'W' command. Be sure to include the buffer terminator, hex Ø.

Restoring a previously saved buffer is as easy as reading the sectors back into memory with the 'R' command. If the command sequence is of lasting importance, you may want to create a binary load file by going to DOS and doing a BINARY SAVE on the command buffer. The 'G' command may then be used to fetch it at any time. Now, say the command buffer starts at $6ØØ. Getting it to automatically execute when it is loaded can be accomplished by appending the following 2 load vectors: 9E ØØ 9F ØØ ØØ Ø6 21 Ø3 22 Ø3 A1 CF. These load vectors simulate the 'O 6ØØ' command by storing the start address at COMPTR and installing the special E: handler (to be described in the next section). In fact, this is a general technique which can be applied to any program which uses the screen editor (E:) for its input. A simple example would be to create an auto-execute command file as described above consisting of DOS 2.ØS commands instead of OMNIMON commands. Then use the 'L' option of DOS 2.ØS to load that file. It should automatically execute the commands you stored in the file.

How do you go about appending these load vectors? One easy way to add a few bytes to the end of a file is as follows:

1) Find the last sector of a file by putting OMNIMON into the linked mode, determining the start sector, reading it into memory with the 'R' command and holding down RETURN until 'EOF' is printed out.
2) Determine the last data byte of the sector by looking at the byte count (the last byte of the sector). In the case of a command buffer this should be a hex Ø.
3) Just past the last data byte add the new bytes. Then increase the byte count to reflect the appended bytes and write the sector back out with 'W(return)'.

Operate from program buffer: O addr

This command is used to execute OMNIMON commands stored as ASCII text somewhere in memory (and terminated with a hex Ø). The format is as follows:

O addr - 'addr' is the address of the OMNIMON commands stored as ASCII text.

Upon execution of the 'O' command, the display editor (E:) device vector at location $321 is replaced with a pointer into a special handler in OMNIMON which takes its input from the program buffer (which is pointed to by $98, COMPTR) instead of the keyboard. It will continue to do so until the command interpreter hits a hex Ø in the program buffer, at which point input it will revert back to the keyboard.

One thing you will notice when using the 'O' command is that only the results of the commands are printed, not the commands themselves. If you need to see the commands also, turn on the printer with 'P' prior to executing 'O'. The commands will show up on the hardcopy.

Move Memory: M addrØ addr1 addr2

This command is for moving blocks of code from anywhere in memory to anywhere in RAM. It matters not if the source and destination blocks overlap. The move command is very simple to use. Just supply the following addresses:

addrØ = source start address
addr1 = source end address
addr2 = destination start address

For example, say you have the following code located at $6ØØ (you can use the 'Y' command to enter it):

```
$6ØØ LDA $2FC
$6Ø3 CMP #$FF
$6Ø5 BEQ $6ØØ
$6Ø7 STA $6ØD
$6ØA JMP $6ØE
$6ØD NOP
$6ØE RTS
```

I know this is a dumb program but I wish to make a point. To move the code to $62Ø, type: M 6ØØ 6ØE 62Ø. Now use 'X' to disassemble the code at $62Ø and you will recognize it as the same. Now, unless the code were relocatable, you would not be able to run the code at the new location without adjusting some of the absolute addresses. Which addresses? The ones which reference locations within the address space of the program. The 'STA $6ØD' and 'JMP $6ØE' would both have to be adjusted. That is the purpose of the Relocate command ('N'), to be described next.

Relocate 65Ø2 Code: N addrØ addr1 addr2 (addr3) (addr4)

Relocate will adjust code assembled to run in one location so that it will execute in another. The format is as follows:

addrØ = start of addr reference range to be adjusted
addr1 = end of addr reference range to be adjusted
addr2 = new base addr
addr3 = start addr of code to be adjusted (default: addr2)
addr4 = end addr of code to be adjusted
(default: addr3+[addr1-addrØ])

This command was designed to be very flexible but, because it has so many options, it can be quite confusing. However, since it was also designed to complement the 'M' command, its use usually requires no thinking at all. Specifically, in the example above we used 'M 6ØØ 6ØE 62Ø' to move the code from $6ØØ-6ØE to $62Ø. Because it has absolute memory references to the range of $6ØØ-6ØE, the copy at $62Ø will not execute properly. If we now move the cursor back up to the 'M' command and change the 'M' to an 'N' ('N 6ØØ 6ØE 62Ø') and hit RETURN, the code at $62Ø will be adjusted to run at $62Ø. Try it and then disassemble the code at $62Ø. Notice the absolute memory references to the range of $6ØØ-6ØE have been adjusted to reflect the new base address of $62Ø.

Now, say we wanted to adjust the code at $600-60E so that it will run at $700 but we don't want to move there to do it (perhaps you did not want to wipe out DOS, which starts at $700). In this case we must specify addr3 and addr4 as follows:

```
N 600 60E 700 600 60E
```

Notice that addr0,1 & 2 are the same as if you had moved the code to $700 first ('M 600 60E 700') but that addr3 & 4 tell 'N' that the code physically resides at $600-60E. After execution of the above command the code at $600-60E disassembles to this:

```
$600 LDA $2FC
$603 CMP #$FF
$605 BEQ $600
$607 STA $70D
$60A JMP $70E
$60D NOP
$60E RTS
```

Notice that the reference to $2FC is not modified because it is outside the address range specified by addr0,1. Also, don't worry about 'BEQ $600' because it is relocatable.

Before you try relocating a big program like a cartridge, let me point out that it will probably still take considerable work. The 'N' command cannot differentiate between 6502 instructions and imbedded program data. The only thing it can do is stop if it hits an illegal opcode and this may or may not be soon enough to avoid modifying some data it should not have (data that happened to look like 6502 code with absolute address references). It also cannot do anything about indirect references to the address range of addr0,1 (for example, indirect references to a data table imbedded within the program). Likewise, jump tables will not be adjusted. The data tables should be easy to locate and move with the 'M' command. The indirect references will probably be more difficult to find and adjust. The jump tables will have to be adjusted by hand. Thus, the 'N' command is most easily used to move relatively small routines around.

Although big programs may present a problem, I will give one other example to show how versatile the Relocate command really is. Say you wanted to move a routine from one part of a program to another. Not only would you have to relocate the routine but you would also have to use the 'N' command on the rest of the program to adjust any references to that routine. Say you had the following program:

```
$600 LDA $604,X (start of routine)
$603 RTS
$604-607        (imbedded data table)
$608 LDX #3     (this is start of program)
$60A JSR $600
$60D STA $D001,X
$610 DEX
$611 BNE $60A
$613 RTS
```

You would like to move the routine and associated data table from $600 to $614. The following commands will do this:

```
>M 600 607 614 (move little routine and table)
>N 600 607 614 608 617 (relocate entire program)
```

After these commands the program should look like this:

```
$608 LDX #3     (this is start of program)
$60A JSR $614
$60D STA $D001,X
$610 DEX
$611 BNE $60A
$613 RTS
$614 LDA $618,X (start of routine)
$617 RTS
$618-61B        (imbedded data table)
```

Study this example carefully. If you understand it completely then you have mastered the 'N' command. You will find it (along with 'Y', 'M' and 'X') a big help in patching programs for which you don't have source code or don't want to take the time to reassemble.

Hexadecimal Arithmetic: H # (# oper) (# oper) ...

This is an enhanced version of the original 'H' command for converting hex to decimal and vice versa. It allows you to specify an additional hex number followed by an operand. The operands are + (add), - (subtract), * (multiply), and / (divide). For example:

```
>H C000 A000 - 80 /
$C000=49152
$2000=2048
$0040=64
>W1 A000 40
```

This example calculates the number of single density sectors required to write an 8K block of memory to disk. $C000 is first converted to decimal, then $A000 is subtracted and the difference is printed out, and finally the result is divided by $80 and the quotient is printed out.

The rules of the Hexadecimal Arithmetic command are:

1) The first number may be in hex or decimal with a decimal number terminated by a non-hex character (i.e., 256T=$100).

2) Every number after the first must be in hex. If you need to use a decimal number, convert it to hex first.

3) The divide operand (/) rounds down for fractions less than .5 and up for fractions greater than .5.

Verify Memory: V addrØ addr1 addr2

Compare 2 blocks of memory and print out the differences. The first block starts with addrØ and ends with addr1. The second block starts with addr2.

P Command Enhancement

The trace turned on by the P command can now be redirected to any output device. If it is desired to output to something other than the printer, store the device specification someplace in memory and point $125 (PBUFAD) to that location. The P command will then open up an I/O channel to that device and the next P command will close it.

For example, if you were to store 'D:TEMP ' (notice the blank used as a terminator) at $6ØØ and store ØØ Ø6 at $125, the P command would open up a disk file (assuming of course that an FMS is in memory). This particular example may cause you some problems if you have a Ramdisk in the system and you are using the 8K banked switch version of OMNIMON. Since the Ramdisk handlers exist only in the lower 4K (L), you would not be able to switch to the upper 4K (U) unless the I/O channel were to a real disk drive whose drive # is greater than that of the Ramdisk.

Sector I/O Enhancements

The buffer default of $6ØØØ no longer exists so you must specify the buffer address on the first sector I/O (R or W). Also, on multiple sector reads you can now get the sector map by holding down the OPTION switch during the operation.

If a Happy drive is read or written to with a sector # of $8ØØ or greater, the Happy drive treats the sector # as an internal buffer address ($8ØØ-$13FF). On multiple reads or writes the sector # is incremented by $8Ø each sector. Usage of the RAM buffer is as follows (compliments of David Milligan of Surrealistic Software):

$8ØØ-$A7F - RAM area for program uploading, as in programming the drive to do something.
$A8Ø-$AFF - RAM area used by the onboard O.S. as a scratchpad area. Used for pointers, flags, etc.
$BØØ-$13FF -RAM area for track reads and writes.

New Ways to Exit OMNIMON

The 'J (addr)' command works as it always has except that if you omit the address it will exit OMNIMON just like START/RETURN. This is to make it convenient to execute a subroutine from the upper 4K (U). The formal way to exit OMNIMON is still START/RETURN and must be executed from the lower 4K.

The 'Z' command exits OMNIMON with an RTS. If you enter OMNIMON via JSR NXTCMD ($CF6C), this is how to exit. The CPU registers are not valid if you enter this way.

OMNIMON User Extensions

Advanced users of OMNIMON may desire to add their own commands to the monitor. The 'L' version of OMNIMON allows you to do just that by providing fixed entry points into the monitor for access to the common I/O routines and a 'U' command for executing a user written extension to the monitor. The extension could be permanently installed into the OS in place of the cassette handlers (assuming you have an EPROM burner) or any one of several could be loaded in from disk with the 'G' (binary load) command. If the location in memory of the extension happens to conflict with the work you are doing at the time, you can move it with the 'M' command and, if necessary, relocate it with the 'N' command (which resides in the 'U' version of OMNIMON). You can see that the user extendability feature of OMNIMONL is indeed very powerful in the hands of an expert!

The first order of business is to explain what resources are available to you as an OMNIMON programmer. The following is a list of the memory locations which are used by OMNIMON:

```
TIBPTR EQU $92 ;POINTER INTO TERMINAL INPUT BUFFER (TIB)
TIBNUM EQU $94 ;LAST # CONVERTED FROM TIB, GEN PURPOSE REG
PARMØ  EQU $96 ;2 BYTE GENERAL PURPOSE REGISTER
FLAGS  EQU $98 ;INTERNAL FLAGS
LAST   EQU $99 ;LAST PERSISTANT COMMAND EXECUTED
XSAVE  EQU $9A ;USED TO SAVE X REGISTER
YSAVE  EQU $9B ;USED TO SAVE Y REGISTER
TEMP   EQU $9C ;2 BYTE GENERAL PUPOSE REGISTER
COMPTR EQU $9E ;2 BYTE POINTER INTO PROGRAM BUFFER
SSFLAG EQU $12Ø ;SINGLE STEP FLAGS
PCSAVE EQU $121 ;USED TO SAVE PC DURING SINGLE STEP
DRVNUM EQU $123 ;CURRENT DRIVE # SELECTED
LINK   EQU $124 ;LINKED OR SEQUENTIAL MODE FLAG
PBUFAD EQU $125 ;POINTER TO DEVICE SPEC WHEN USING 'P' CMND
USRVEC EQU $127 ;POINTER TO USER EXTENSION ROUTINE
TIB    EQU $129 ;TERMINAL INPUT BUFFER
```

These are the locations used by the L and H versions. Every attempt will be made to keep these variables fixed from now on. These may or may not be correct for earlier versions of OMNIMON. To be sure, scan the OMNIMON code starting at $CØØØ and it should be easy to determine which part of the zero page and the stack are used for these variables. It is advisable to use the same labels (TIBPTR, TIBNUM, etc.) in your programming.

By the way, the zero page locations used by OMNIMON are preserved on the stack upon entry to the monitor and restored on exit. Thus, a routine you are debugging with OMNIMON may use these zero page locations, even when using the 'J' or 'E' command to execute them from OMNIMON. (Did you know that both J and E leave OMNIMON while executing your code and return upon completion?) However, they will not reflect the correct values if you use the 'D' command to look at them because the OMNIMON variables are in use at that time.

OMNIMON Entry Points

There are quite a number of routines in OMNIMON which will be useful to you as a programmer. Access to these routines is provided in OMNIMONL by a number of fixed entry points, 'fixed' meaning that these addresses are guaranteed to remain the same in future revisions of OMNIMONL. Some of these routines help you process user input, while others allow you to format data output and messages to the screen. Still others help you interface to the disk drive. Each entry point will be covered in detail, but first let's take a look at what is available and a short example.

Subroutine Calls:

PCTPAR	$CF1E	Get user program counter to PARMØ
PRTLIN	$CF21	Print 8 bytes @PARMØ in current output format
INSNUM	$CF24	Install Ramdisk as drive specified in TIBNUM
DECONV	$CF27	Convert user decimal input to hex in TIBNUM
DRVTYP	$CF2A	Read drive status to determine density
SECTRD	$CF2D	Read a sector from disk into memory
SECTWR	$CF3Ø	Write a sector to disk from memory
SECINC	$CF33	Increment sector # and buffer address
TIBCNV	$CF36	Convert user hex input to hex in TIBNUM
PRTHEX	$CF39	Print byte in accumulator to screen in hex
PRTCPU	$CF3C	Print contents of CPU registers to the screen
ARGU3	$CF3F	Fetch 3 user arguments
NUMTOP	$CF42	Move 2 byte contents of TIBNUM to PARMØ
INCPTR	$CF45	Increment 2 byte pointer,TIBPTR, by 1
PRTADR	$CF48	Print 2 bytes in PARMØ to screen in hex
GETREC	$CF4B	Get a line of input from the user -> TIB
INCPAR	$CF4E	Increment 2 byte pointer, PARMØ, by 1
PRTBLK	$CF51	Print # blanks to screen as specified in ACC
HEXCON	$CF54	Convert ASCII in accumulator to hex
SSGET	$CF57	Fetch 2 user arguments
PRINT	$CF5A	Print inline text to screen
PUTCHR	$CF5D	Print ASCII character in accumulator to screen
PRTCHR	$CF6Ø	Force (unprintable) ASCII character to screen
CIOCHR	$CF63	Pass character in accumulator to CIO

Jump Entry Points:

BOOT	$CF66	Boot from the drive specified in DRVNUM
INPERR	$CF69	Return from command which prints 'INPUT ERR'
AUTGO	$CF1B	Execute from command buffer starting @COMPTR
NXTCMD	$CF6C	Normal return from command
RESTOR	$CF6F	Normal exit from OMNIMON (restores zpage vars)
RESET	$CF72	Normal RESET
RESET2	$CF75	Altered RESET (SELECT/RESET)
PWRUPØ	$CF78	Powerup entry for lockup recovery

Notice that the entry points of the first set are called with 'JSR' while those of the second set are used with 'JMP'. Once again, it is advisable to use the mnemonic labels provided here rather than make up your own. This will make it easier for other OMNIMON users to understand your routines. In fact, you are invited to submit your OMNIMON extensions to CDY for inclusion in an 'OMNIMON User Extentions' disk.

OMNIMON Extension Example

A short example might help you understand why it is that you would want to write an extension to OMNIMON. Say you are developing a program for publication and you are working on a protection scheme. Part of a typical protection scheme it to use a byte as a mask to EOR (exclusive-or) over a range of memory and then to do a checksum over the range. To facilitate devolopment you wish to have an OMNIMON command which will accept an address range and a mask byte as arguments and then do the EOR and checksum over the specified range. The following OMNIMON extension will do the job:

```
TIBPTR EQU  $92
TIBNUM EQU  $94
PARMØ  EQU  $96
USRVEC EQU  $127       ;POINTER TO USER EXTENSION
TIBCNV EQU  $CF36      ;CONVERT A PARM FROM USER LINE TO HEX
NUMTOP EQU  $CF42      ;TRANSFER 2 BYTES OF TIBNUM TO PARMØ
INCPAR EQU  $CF4E      ;INC 2 BYTE POINTER, PARMØ, BY 1
PRINT  EQU  $CF5A      ;PRINT INLINE TEXT TO SCREEN
PRTHEX EQU  $CF39      ;PRINT ACCUMULATOR TO SCREEN IN HEX
PUTCHR EQU  $CF5D      ;PRINT CHARACTER IN ACC TO SCREEN
NXTCMD EQU  $CF6C      ;EXIT USER EXTENSION

  ORG  USRVEC     ;THIS LOAD VECTOR WILL INSTALL ROUTINE
  DW   EORMEM

  ORG  $6ØØ
*
* EOR MEMORY ('U start-addr end-addr mask-byte')
*
EORMEM
  JSR  TIBCNV     ;FETCH START ADDR FROM USER LINE -> TIBNUM
  BMI  EOREX      ;DID USER NOT SUPPLY A START ADDRESS?

  JSR  NUMTOP     ;TIBNUM -> PARMØ
  JSR  TIBCNV     ;FETCH END ADDR FROM USER LINE -> TIBNUM
  BMI  EOREX      ;DID USER NOT SUPPLY AN END ADDRESS?

  LDA  TIBNUM     ;PUSH END ADDR ONTO STACK
  PHA
  LDA  TIBNUM+1
  PHA
  JSR  TIBCNV     ;GET MASK BYTE FROM USER LINE -> TIBNUM
  BPL  EORMSK     ;DID USER SPECIFY A MASK?

  LDA  #$FF       ;NO, ASSUME #$FF
  STA  TIBNUM
EORMSK
  LDY  TIBNUM     ;MASK -> Y
  PLA             ;RESTORE END ADDR TO TIBNUM
  STA  TIBNUM+1
  PLA
  STA  TIBNUM
```

```
      TYA
      PHA               ;PUSH MASK ONTO STACK
      LDY  #Ø
      STY  TIBPTR       ;USE TIBPTR TO ACCUMULATE CHECKSUM
    EORUP
      PLA               ;GET MASK
      PHA
      EOR  (PARMØ),Y    ;MASK BYTE OF MEMORY
      STA  (PARMØ),Y    ;STORE THE MASKED BYTE
      CLC
      ADC  TIBPTR       ;ACCUMULATE CHECKSUM
      STA  TIBPTR
      JSR  INCPAR       ;INCREMENT START ADDR BY 1
      SEC               ;END ADDR - START ADDR
      LDA  TIBNUM
      SBC  PARMØ
      LDA  TIBNUM+1
      SBC  PARMØ+1
      BPL  EORUP        ;NOT DONE ENTIRE RANGE?

      PLA
      LDA  TIBPTR
      PHA
      JSR  PRINT        ;INSERT A LINE AND PRINT MESSAGE
      DB   $9D,'CHECKSUM = ',Ø
      PLA
      JSR  PRTHEX       ;PRINT CHECKSUM
      LDA  #$9B
      JSR  PRTCHR       ;PRINT RETURN
 EOREX                  ;EXIT TO OMNIMON PROPER
      JMP  NXTCMD

DUMMY                   ;DUMMY RTS FOR ATARI MACRO-ASSEMBLER AUTORUN
      RTS
      END  DUMMY
```

This routine contains all the elements of a typical OMNIMON extension. First you have the EQUates of all the OMNIMON resources you will use in the routine. Then you have the load vector to install the routine by storing the address of the routine at USRVEC. (The ATARI Macro-Assembler creates a new load vector each time an 'ORG' is used. If your assembler does not do this then you must do it by hand:

```
                        27 Ø1    DW USRVEC
                        28 Ø1    DW USRVEC+1
                        ØØ Ø6    DW EORMEM
```

If you do not understand about load vectors then read the section entitled 'BINARY LOAD FILES'.) The routine itself starts by reading the 3 user input parameters (JSR TIBCNV) and storing them in PARMØ (start addr), TIBNUM (end addr), and the accumulator (mask). Then the processing is done with the checksum being accumulated in TIBPTR. Finally, the checksum is printed out and the user extention is terminated with 'JMP NXTCMD' to return to OMNIMON proper. Study this routine carefully and then read the detailed descriptions of the entry points which follow.

User Input

TIBCNV($CF36): Converts the next hex # in the user input line to 2 bytes in TIBNUM.
TIBPTR: Points at or before next # to be converted in TIB
JSR TIBCNV -> Affects ACC, X, YSAVE, TIBNUM, TIBPTR
TIBNUM: Holds 2 byte result of conversion
N Flag: Set if conversion unsuccessful

GETREC($CF4B): Calls CIO to get next user input line to TIB. Resets TIBPTR beginning of TIB. Puts line to printer if open.
JSR GETREC -> Affects ACC, XSAVE, YSAVE, TIBPTR, COMPTR
TIBPTR: Points to beginning of TIB

HEXCON($CF54): Converts ASCII in accumulator to hex in ACC.
ACC = ASCII # to convert ($30-39, $41-46)
JSR HEXCON -> Affects ACC
ACC = Hex byte ($00-0F)
N Flag: Set if conversion unsuccessful

DECONV($CF27): Converts decimal # in TIB to 2 bytes in TIBNUM.
TIBPTR: Points to the beginning of decimal # in TIB
ACC = First character of decimal # in TIB
Y REG = 0
JSR DECONV -> Affects ACC, X, TIBPTR, TIBNUM, FR0 ($D4-D9)
TIBNUM: Holds 2 byte result of conversion
TIBPTR: Points just past decimal # in TIB

ARGU3($CF3F): Converts 3 hex input parameters of the form: start-addr end-addr destination-addr. Used in 'M', 'N' and 'V'.
TIBPTR: Points to first hex # in TIB
JSR ARGU3 -> Affects ACC, X, YSAVE, TIBPTR, TIBNUM, PARM0, TIB
PARM0 : start address
TIBNUM: destination address
TIB, TIB+1: end address - start address (# bytes)
N Flag: Set if conversion unsuccessful

SSGET($CF57): Converts 2 hex input parameters of the form: start-addr end-addr. Assumes defaults if one or both addrs missing.
TIBPTR: Points to first hex # in TIB
JSR SSGET -> Affects ACC, X, YSAVE, TIBPTR, TIBNUM, PARM0
PARM0 : Start address (unchanged if parameter missing)
TIBNUM: End address (=PARM0 if parameter missing)

Screen Output

PRINT($CF5A): Prints inline ASCII text following JSR PRINT and terminated by 0.
JSR PRINT -> Affects ACC, XSAVE, YSAVE, TIBPTR

PRTBLK($CF51): Print the # of blanks specified in ACC
ACC : # blanks you wish to print (1-255)
JSR BLANK -> Affects ACC, TEMP, XSAVE, YSAVE

PUTCHR($CF5D): Print ASCII character in ACC to screen and echo to the printer if enabled.
ACC : ASCII character
JSR PUTCHR -> Affects XSAVE, YSAVE

PRTCHR($CF60): Force (unprintable) char to screen, translate to printable char (-) if necessary, and echo to printer.
ACC : ASCII character (e.g., $9B can be printed this way)
JSR PRTCHR -> Affects XSAVE, YSAVE

CIOCHR($CF63): Pass byte in ACC to CIO.
ACC : byte to pass
X REG : IOCB# * $10 of an OPEN channel
JSR CIOCHR -> Affects YSAVE

PRTADR($CF48): Print 2 bytes of PARM0 in hex
PARM0 : 2 byte # to print
JSR PRTADR -> Affects ACC, XSAVE, YSAVE

PRTHEX($CF39): Print byte in ACC in hex
ACC : Byte to be printed
JSR PRTHEX -> Affects ACC, XSAVE, YSAVE

PRTCPU($CF3C): Print contents of CPU registers
JSR PRTCPU -> Affects ACC, Y, XSAVE, YSAVE, TIBPTR, TIBNUM, PARM0, TEMP

PRTLIN($CF21): Print 8 bytes @PARM0 in current ouput format
PARM0 : Points to 1st of 8 bytes to be printed
JSR PRTLIN -> Affects ACC, Y, XSAVE, YSAVE

Disk Input/Output

INSNUM($CF24): Install Ramdisk handlers into whatever program is in memory (replace all references to $E459 and $E453).
TIBNUM: Drive # to assign to Ramdisk
JSR INSNUM -> ACC, Y, X, TIBPTR, XSAVE, YSAVE, PARM0

DRVTYP($CF2A): Read drive status to determine density and set the byte count in the DCB in anticipation of next READ or WRITE
DRVNUM: Drive # (1-4)
$30A,30B: Sector # to be next read or written
JSR DRVTYP -> Affects ACC, X, Y
$308,309: Byte count=$80 for single density, $100 for double

SECTRD($CF2D): Read a sector from disk into memory. You should JSR DRVTYP just prior to using this for the first time.
$301 : Drive # (set in DRVTYP)
$304-305: Buffer address
$308-309: Byte count (set in DRVTYP)
$30A-30B: Sector # (set prior to JSR DRVTYP)
JSR SECTRD -> Affects ACC, X, Y
N Flag : Set if error, ACC and Y= status

SECTWR($CF30): Write a sector from memory to the disk. You should JSR DRVTYP just prior to using this for the first time.
$301 : Drive # (set in DRVTYP)
$304-305: Buffer address
$308-309: Byte count (set in DRVTYP)
$30A-30B: Sector # (set prior to JSR DRVTYP)
JSR SECTRD -> Affects ACC, X, Y
N Flag : Set if error, ACC and Y= status

SECINC($CF33): Increment DCB sector # by 1 (or $80 if sector # >= $800) and buffer address by the # of bytes/sector.
$308-309: Byte count (set in DRVTYP)
$304-305: Buffer address (left from last READ or WRITE)
JSR SECINC -> ACC
$30A-30B: Sector count incremented by 1
$304-304: Buffer addr incremented by $80 for SD, $100 for DD

Miscellaneous

PCTPAR($CF1E): Get user program counter (PC) to PARM0. There must be exactly 1 return address on the stack when this routine is called. If you wish to call it, push 2 bytes into the stack just prior to the call and pull them off afterwards.
STACK : Must be nested once from OMNIMON proper
JSR PCTPAR -> ACC, Y, XSAVE, TIBNUM, PARM0
PARM0 : User program counter
TIBNUM: Points into stack 1 byte below user CPU registers

NUMTOP($CF42): Transfer 2 byte contents of TIBNUM to PARM0
JSR NUMTOP -> Affects ACC, PARM0
PARM0 : Identical to TIBNUM

INCPTR($CF45): Increment 2 bytes of TIBPTR by 1.
JSR INCPTR -> TIBPTR
TIBPTR: Incremented by 1

INCPAR($CF4E): Increment 2 bytes of PARM0 by 1.
JSR INCPAR -> PARM0
PARM0: Incremented by 1

JMP Entry Points

BOOT($CF66): Boot from the drive specified in DRVNUM.
NXTCMD($CF6C): Normal return from a command
INPERR($CF69): Same as NXTCMD except 'INPUT ERR' printed
AUTGO($CF1B): Execute from command buffer @COMPTR and then JMP NXTCMD (Same as 'O addr' command)
RESTOR($CF6F): Exit from OMNIMON. User CPU state restored.
RESET($CF72): Normal SYSTEM RESET
RESET2($CF75): Altered RESET (SELECT/RESET)
PWRUP0($CF78): Powerup entry for lockup recovery

Extension Suggestions: 1) Search disk for sequence. 2) Search memory with wildcards. 3) A simple line editor for editing disk files under OMNIMON control.

AXLON Ramdisk Support: 'I' and 'B' commands

These commands allow Ramdisk owners to take full advantage of the power and flexibility of this marvelous device. The resident Ramdisk handlers in OMNIMONL allow you to use your Ramdisk with any DOS which uses standard SIO calls ($E459 and $E453). Operation of the Ramdisk in conjunction with drives other than single density is possible if the DOS will support them (e.g., OSA+, MYDOS, etc.). In addition, you will find it possible to use the Ramdisk with other boot programs which require a lot of disk access (e.g., DBMSs, word processors, games, etc.). The general rule is that any program that will restart when you hit SYSTEM RESET (instead of rebooting) should be able to use the Ramdisk just like any other single density disk drive. It also makes things easier if the program has no disk copy protection.

The main command to support Ramdisk is 'I (drive#)'. This command installs the resident Ramdisk handlers into whatever software is currently in memory. It does so by searching all of memory for all references to $E459 and $E453 and replacing them with hooks into OMNIMON ($CFC8 and $CFCB respectively). In this way all SIO calls are intercepted and examined to see if the Ramdisk is being addressed. If it is, the special handlers take over. Else, the call is passed on to SIO.

Installation Technique #1

The basic technique for installing the Ramdisk handlers is to boot-up the software, pop into OMNIMONL with OPTION/RESET, use the 'I' command and then restart the program by holding down the START switch and typing RETURN. This is the same as doing a warmstart with a brief stopover in OMNIMONL. If hitting RESET causes th program to reboot (e.g., ATARI DOS 2.ØS) then you may use SELECT/RESET to interrupt the program. If the program restarts when you exit OMNIMONL with START/RETURN, then everything is probably fine. If you are unable to restart the program without rebooting then another method can be used (see Installation Technique #2).

Let's take a typical example:

1) Be sure that your Ramdisk is enabled and boot up ATARI DOS 2.ØS (or any of the many modified versions). Once the DOS menu appears, pop into OMNIMONL with SELECT/RESET.
2) Now type 'I(return)'. (If you do not specify a drive # after the 'I', drive #1 will be assumed.) Drive #'s equal to or greater than the Ramdisk drive # will be incremented by 1.
3) Return to the DOS menu by holding down the START switch and typing RETURN. Hit RETURN once again to get the menu back.
4) Format the Ramdisk with the I command of DOS.
5) Write DOS files to the Ramdisk with the H command of DOS.
6) Hit SYSTEM RESET. If you did the previous steps correctly, the DOS menu should appear very quickly since it will now boot out of the Ramdisk.

Now you can treat the Ramdisk just like any other single density drive in the system. If you wish to assign it a different drive #, pop into OMNIMONL and use the 'I' command again with the new drive #. Notice that once the 'I' command has been used, the 'R' and 'W' commands of OMNIMONL will also treat the Ramdisk just like another disk drive. Use the 'L(#)' command to address the different drives in the system.

Ramdisk installation can be accomplished from assembly language by storing the drive # in TIBNUM ($94) and doing a JSR INSNUM ($CF24). The following 11 bytes appended to the end of a binary load file will automatically install the Ramdisk handlers (where drv# = 1 to 4): 94 00 94 00 drv# E2 02 E3 02 24 CF. Appended to the end of DUP.SYS, these load vectors will take the place of installation technique #1 when you boot up DOS.

Installation Technique #2

This method will allow you to use Ramdisk with many boot programs which do not use DOS or have their own file management system. This method will work only if the program is on an unprotected disk.

1) Boot up a disk sector copying program. Install the Ramdisk as drive #1 using technique #1.
2) Duplicate the boot disk using the Ramdisk as the destination drive.
3) Pop into OMNIMONL again and select the Ramdisk with the 'L' command. (It is already selected if it is drive #1.)
4) Now we must install the Ramdisk handlers into the program on the Ramdisk. This is done by reading the program into memory, using the 'I' command and writing the program back out to disk. If the program takes the entire disk then the following sequence will work: R1 400 100,I,W1 400 100,R101 400 100,I,W101 400 100,R201 400 D0,I,W201 400 D0. This is not as much typing as it looks because of the screen editing features of OMNIMON.
5) Now type 'B(return)'. This command will boot off the selected drive but most of the time will work only on drive #1. It is especially useful for booting off the Ramdisk.

This method may not work with some programs because they have interrupt routines located in the address range of $4000 to $7FFF. Because Ramdisk uses this area of memory for its bank switching, you can see how an interrupt routine in this region would not work too well during Ramdisk I/O.

Once the program is in the Ramdisk, you can always reboot it with the 'B' command, even if you have run other programs since copying it up there. This is especially true if you do the simple hardware mod described next. But don't cycle power or the contents of Ramdisk will be lost. If you wish to do a coldstart, do a 'JE477 (return) START/RETURN' from OMNIMON or install a coldstart switch as indicated on the next page. By the way, the Ramdisk uses banks 1 to 7 while the user bank is bank 0.

Ramdisk Hardware Modification

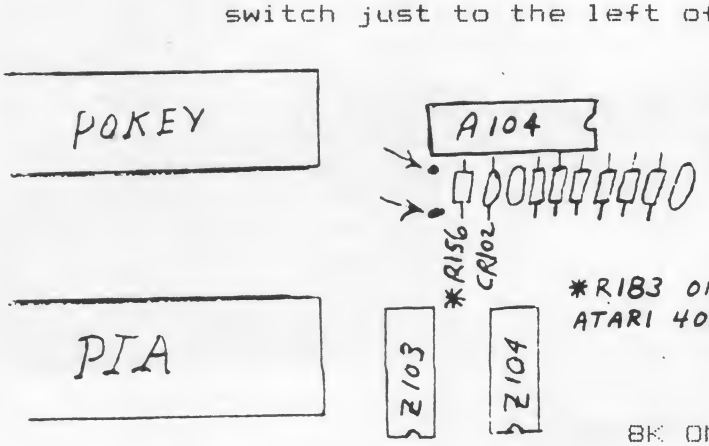
A very useful modification can be made to your Ramdisk board to make it much better behaved. It will save you from having to flip the switch on the Ramdisk in order to run software which loads into the $FCØ-$FFF range. As it stands, writing to this address range will cause Ramdisk banks to be switched in in place of user memory. This usually causes the program to crash. This mod disables the $FCØ-$FFF range and doubles the other select range from $CFCØ-$CFFF to $CF8Ø-$CFFF, which should cause no problem. If you do this mod carefully it will probably not affect the warranty but it is best to check with AXLON to be safe. In the past they have recommended it.

1) Locate pin 18 on the card edge of the RAM board in the front RAM slot (the slot just behind the OS board). It does not matter what type of RAM board this is (16K or 32K). Follow the etch back to a convenient place to solder (perhaps the pin of an IC) and solder one end of a 6" wire to this spot. On an ATARI 16K board this would be Z5Ø1 pin 1.
2) Locate the 74LS133 IC on the Ramdisk. Solder the other end of the wire to pin 15 of this IC. Referring to the diagram below, make the indicated cut on the underside of the Ramdisk board and the mod is complete.

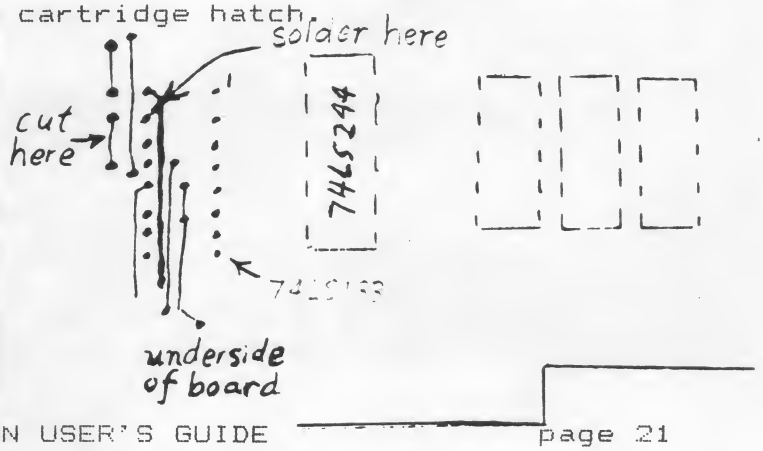
Coldstart Switch

This switch connects the RESET button directly to the Reset pin on the 65Ø2, allowing you to do a coldstart without cycling power. This is especially useful in conjunction with the Ramdisk because its contents are preserved as long as you don't cycle power. (This is not true under the original Memory Management System supplied by AXLON.) It is also possible to recover from system lockup by pushing the coldstart switch and popping into OMNIMONL with SELECT/RESET. From that point you can many times restart the program by hitting SYSTEM/RESET (e.g., BASIC).

Installation is accomplished by disassembling the computer and locating the pads on the motherboard as indicated in the drawing for the 8ØØ. On a 4ØØ the pads are located in the back right hand corner between R179 and CR1Ø3. A spring loaded switch (cheap Radio Shack pushbutton is fine) should be put in series with a 47 Ohm 1/4 watt resistor between the two pads. Mount the switch just to the left of the cartridge hatch.



800 Reset mod



Ramdisk Mod

A: Alter Memory: A addr byte byte ... - Used to change 1 or more contiguous bytes of memory.

B: Boot Disk: B - Will boot off of the selected drive. Especially useful in conjunction with the AXLON Ramdisk.

C: CPU Registers: C - Used to display and alter the registers.

D: Display Memory: D (start addr) (end addr) - Used to view memory data. To alter memory, position cursor and type change.

E: Execute Memory: E (option/# steps) - Will execute one or more instructions at a time and display intermediate results.

F: Fill Prgm Buffer: F addr - Teach monitor a sequence of commands for later execution with the 'O' command.

G: Get File: G (file spec) (addr) - A full binary load function, single or double density. Doubles as disk directory command.

H: Hex Conversion: H # - Allows conversion from hex to decimal and vice versa on numbers of 1 or 2 bytes.

H: Hex Arithmetic: H # (# oper) (# oper) ... - Hex conversion allowing addition, subtraction, multiplication and division.

I: Install Ramdisk: I (drv#) - Install Ramdisk handlers into whatever program is resident in memory at the time.

J: Jump Subroutine: J (addr) - Go execute subroutine.

L: Link Drive: L (drv#) - Select drive # and linked or sequential sector modes. All disk I/O will go to the selected drive.

M: Move Memory: M addrØ addr1 addr2 - Move a block of memory from anywhere in memory to anywhere else.

N: Relocate Memory: N addrØ addr1 addr2 (addr3) (addr4) - Adjust 6502 code that it will execute in another location.

O: Operate Prgm Buffer: O addr - Execute the commands stored earlier with the 'F' command.

P: Printer Control: P - Screen I/O can be echoed to a printer.

R: Read Disk: R (sect#) (buff addr) (# sects) - Read one or more sectors from selected disk drive into a specified buffer area.

S: Search Memory: S addr byte byte ... - Search memory for and print out occurrences of a sequence of bytes.

T: Toggle Format: T - Toggle between hex and character formats.

U: User command: U - Allows access to user written extensions of OMNIMON. Entry points for commonly used routines provided.

V: Verify Memory: V addrØ addr1 addr2 - Compare 2 blocks of memory and print out the differences.

W: Write Disk: W (sect#) (buff addr) (# sects) - Write one or more sectors to disk from a specified buffer address.

X: Disassembler: X (addrØ) (addr1) - Translate machine code into assembly language. Can be used to create a source file.

Y: Assembler: Y addr instr - Translate assembly language into machine code one line at a time. Useful for patching programs.

Z: Exit Monitor: Z - Special exit for use with 'O' command.

Lockup Recovery - Allows recovery from system lockup without destroying contents of memory. Requires simple hardware mod.

I/O Redirection - Allows printer I/O to be redirected to any device, including a disk file. Saves lots of paper!

Happy Support - Allows for easy transfer of memory to and from the RAM buffers in the Happy drive.

Bit-3 Support - Allows user to turn on and off the 80 column mode of the Bit-3 board from OMNIMON.